1982

Should Distributed Systems be Hidden?

Peter J. Denning

Robert L. Brown

Report Number:
82-426

SHOULD DISTRIBUTED SYSTEMS BE HIDDEN?

Peter J. Denning[1]
Robert L. Brown[1]

Computer Sciences Department
Purdue University

Abstract: Existing multiprocessor systems can be divided in three classes according to the degree of coupling among the machines: shared memory, local network, or long-haul network. The tighter the coupling, the deeper in the operating system will be the optimal position of the software for interprocess communications. Current designs attempt either to extend shared memory operating systems, or long-haul network protocols, to the local network. Neither approach is optimal. We argue that the best position of the communications layer is in the middle levels of the operating system -- above the virtual memory and below the directory manager. With this position, the directory hierarchy can be extended to become a global name space for permanent objects in the system. Each of the higher levels is then easily converted to hide the remaining vestiges of the locations of objects it manages. The resulting operating system can fit on machines as small as workstations.

December 1982
CSD-TR-426

This is a preprint of a paper to be presented at the International Workshop on Computer Systems Organization, March 29-31, 1983.

---

[1] Authors addresses: Peter J. Denning, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: denning@purdue); Robert L. Brown, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: rlb@purdue).

INTRODUCTION

Distributed systems have been under active discussion for a decade. A few have been in operation for nearly that span, others are being delivered, and many more are being designed. It is fitting to reflect on what has been accomplished.

In this paper *distributed system* means a coordinated collection of computers connected by a communication network.[2] The computers and network are intended to form a system with personalized working environments, easy communication among users, easy access to special services such as mail and printing, continued operation despite failures of individual computers, and the ability to extend the power of the system in modest steps by adding new computers. Are these goals being met by current designs?

In many cases, no. Some distributed operating systems are nothing more than traditional designs with long-haul network software added on to manage communication with other machines. By allowing the machines and network to be visible in the programming environment, these systems force many users to learn about transport-level network protocols to carry out ordinary tasks. Their programming environments are lower in level than those of many third generation operating systems. Other distributed operating systems are nothing more than a traditional design adapted by putting important functions on dedicated machines. These systems hide the machines and the network but, because functions are not replicated, they are no more fault tolerant than the single-machine

---

[2] Sometimes the term is used to denote an array of microcomputers that work in parallel toward the solution of a common problem. (For example, [Brin78, Hoar78].) This class of distributed systems is not considered here.

operating systems from which they were derived.

The goals of distributed system cannot be met without a redesign of the operating system's architecture. This paper will argue that a layered operating system is straightforward to extend for computations among several machines on a local network. The existence of multiple computers and a network will not be observable in the resulting user environment; the system will have a high degree of functional redundancy. Other approaches to this problem appear in Tandem systems [Bart81] and the Grapevine system [Birr82]. The UCLA LOCUS system [PopW81] addresses some of the issues handled here in a strictly UNIX environment.

The key principles of our proposal are: process-to-process communication can be implemented in a middle layer of the operating system; a global name space for permanent objects can be implemented with distributed directories; and all global objects can be accessible through a uniform interface. These ideas will be illustrated with specifications of the external interfaces implemented at the communications, directories, and higher levels of the operating system.

DISTRIBUTED SYSTEMS

An important difference between a single-machine system and a multi-machine system is in the mechanism for exchanging information between processes. In a conventional operating system, the cost of sending a message need not exceed the fixed cost of sending the pointer for the memory segment

containing the message. In a distributed system, however, the cost of sending the message depends on the size of the message, and on the mechanisms for recovery from network errors, if the two processes are on different machines.

The requirements of reliable, efficient message exchange strongly influence the placement of the communications protocols within an operating system. They therefore affect the degree to which dependence on physical factors can be hidden from the user.

System Types

Multiprocessor systems can usually be placed in one of three classes according to the type of communication medium: shared memory, local computer networks, and long-haul networks. We are primarily interested in local network systems in this paper.

Shared memory systems, such as C.mmp [Wulf81] and Intel 432 [Inte81, Kahn81, Cox81, Poll81], consist of several processors connected by a bus or crossbar switch to memory. A global scheduler determines an assignment of ready tasks to the processors. Messages can be passed from one process to another at unit cost; they can be copied at the saturation rate of the memory.

Local computer networks, such as Cm* [Swan77, Jone79, Oust80], Xerox Star [Smit82], or the Cambridge Ring [Wilk79], consist of several processors connected by wide-band serial communications lines. A global scheduler, or small set of coordinated schedulers, assigns ready tasks to processors. Some processors may have preassigned functions, such as file server, printer server, login server, mail server, or workstation. Message passing cost is proportional to

message size. Transmission speeds vary from 4 Mbits/sec up to memory speed. In practice, these networks are highly reliable: few retransmissions result from garbled packets.

Long haul networks, such as ARPANET [Post81, Tane81], consist of many computer systems, typically of different types, connected by narrow band serial communications lines. There is no global scheduler. Message passing cost is proportional to message size and error rates in the network. Transmission speeds range from 300 bits/sec up to 100K bits/sec. Errors in the network are common.

The placement of multiprocessor support in an operating system depends on the medium for interprocess communication. It can be placed at a very low level in a shared memory system because the degree of information sharing is high, the granularity is low, message exchange is cheap, and transmission errors are rare. (See [Denn81].) It must be placed at a high level in long-haul systems because the computers are too heterogeneous and do not have common standards for sharing and message passing. Multiprocess support is best placed at the middle levels of an operating system for local networks. (This statement will be made precise in the next section.)

The first problem cited in the introduction arises because the software originally designed for long-haul networks has been used for local networks even though it is not optimized for this application. This is primarily a product of expediency: long-haul software has existed longer and is available widely. Similarly, the software developed for shared memory systems is not appropriate for local networks because it has been optimized for unit-cost, error-free message exchange.

Notable examples of local networks implemented with long-haul software are found among UNIX systems (e.g., 4.2bsd [Joy82]). In these cases, the network and its protocols are visible in the user environment. The user becomes a party to major decisions involving the physical resources of the system, such as the machine selection problem, the file location problem, and the intermachine naming problem. The user may, therefore, have considerable difficulty accomplishing ordinary tasks. For example, the user may have to explicitly invoke remote directory searches and file transfers if the required files are on different machines. (This can happen if he should log in on a different machine from his prior session or if the last person to work on shared files moved them.) He may have to use remote login to gain access to a service provided by another machine. He may have to use sequences of machine names as addresses for mail or other users. Besides the availability of the software, the primary advantage of this approach is its functional redundancy – all the functions appear on every machine and, hence, the failure of one machine will not bring down the entire network. The primary disadvantage is the appearance of networks in the user environment.

Another group of distributed operating systems for local networks are adaptations of shared memory operating systems. Notable examples include StarOS [Jone79], Medusa [Oust80], the Xerox Star system [Smit82], and the Cambridge Ring system [Wilk79]. The shared memory operating system has been adapted for communication over the local net; this is accomplished by implementing important systems functions as dedicated processes that are invoked by sending them messages and awaiting responses. In StarOS, processes are assigned to specific processors by a small set of coordinated schedulers. In the Cambridge

Ring some processes -- such as the file server, the printer server, and the login server -- are preassigned to specific machines; user processes are assigned to idle minicomputers by the login server. In Xerox Star, all processes are preassigned to machines. Adapting the architecture of a shared memory operating system for the local network expedites development because it requires few new concepts. It produces a uniform user environment that hides the network. The primary disadvantage is the lack of functional redundancy inherited from the single-machine operating system. The entire system may fail if a single machine fails -- e.g., if the file server breaks down.

Adapting a shared memory operating system or long-haul network software to a local network does not produce the best results. The solution requires a redesign of the operating system so that the resulting network of machines is hidden yet efficient. We will argue in the next section that the design effort can be straightforwardly managed in the context of a multilevel design hierarchy.

OPERATING SYSTEMS

Operating systems perform two classes of functions: allocating resources among computing tasks and extending primitive hardware by implementing a powerful virtual machine that serves as a high-level programming environment. The second function has evolved from the methods developed to implement the first.

The principle of data abstraction -- hiding away the details of managing a class of objects inside a module that has a simple, high-level interface with its

users -- was recognized early as an essential tool for maintaining consistency of resource allocation state information. (See [Denn66].) This principle has been extended to an integrated view of an operating system as a hierarchy of abstract machines from the base hardware to the user interface. The first instance of a multilevel operating system was reported by Dijkstra in 1968 [Dijk68]. The Provably Secure Operating System (PSOS) is the first complete layered system reported and formally proved correct in the open literature. [Neum80]

Table I is an example of a 15-layer operating system design. This design incorporates the principles of UNIX [Ritc74] in a framework like PSOS [Neum80].

Each level in Table I manages a set of objects of given type; it does this by providing operations for creating, deleting, and changing the states of objects. For example, Level 5 implements primitive (sequential) processes and semaphores and hides the details of processor scheduling and interrupts; its interface to the higher levels includes the **wait** and **signal** operations. Level 8 provides an interface to secondary units, such as disks, for long term storage of objects created by the higher levels. Level 13 implements user processes -- i.e., the virtual machines containing user programs in execution. (A user process contains a primitive process, virtual memory, a current directory pointer, and parameters passed on its invocation.) The lower-numbered levels are the most primitive and are masked from view by the higher levels. For example, instructions that lock and unlock a bit in a memory word are no longer visible above the Process Manager level. The programs implementing operations at a given level may call operations of lower levels but not of higher levels. For example, the file system may use semaphores to lock shared data during update, but the Process Manager level cannot store information in files.

HIDING DISTRIBUTIVITY

It is obviously impossible hide all aspects of a distributed system. Some sites on a network have special designations associated with their locations – e.g., a "New York City sales database." The response time of an open file command is obviously shorter if the file is stored locally than if the file must be moved from another machine.

Hiding the Bindings

Hiding the maps from names to objects is a central principle in many third generation operating systems. It is responsible for the machine independence of the user environment. In multilevel systems, the map for a given class of objects is maintained by the layer for that class.

To carry this principle over to multimachine systems, the design must hide the locations of all sharable objects (directories, files, pipes, devices, user processes, and extended type objects). This requires the solution of two problems: the reliable exchange of information between processes on different machines and global naming of objects. The first problem is solved by the *communications layer*, the second by a *common directory tree hierarchy* among the various machines. These problems will be considered in the next two subsections.

The communications layer can be inserted in the middle of the design hierarchy, between the local long-term store level and the directory level. The resulting set of levels is illustrated in Table II. If the communications layer were higher than the directory level, the directory manager would have no access to

Level	Name	Objects	Example Operations
1	Electronic Circuits	Registers, gates, busses, etc.	clear, transfer, complement, activate, etc.
2	Instruction Set	Evaluation stack, microprogram interpreter	load, store, un_op, bin_op, branch, array_index, etc.
3	Procedures	Procedure segments, Call stack, display	mark, call, return
4	Interrupts	Interrupt and fault handler programs	invoke, mask, unmask
5	Primitive Processes	Primitive process, semaphores, ready list	suspend, resume, wait, signal
6	Capabilities	Capabilities, domains	make, copy, amplify, enter, verify parameters
7	Local Secondary Store	Blocks of data, disk device drivers	read, write, allocate, free
8	Virtual Memory	segments	read, write, fetch
9	Directories	Directories	create, attach, detach, search, list
10	File System	Files	open, close, read, write
11	Pipes	Interprocess data pipes	open, close, read, write
12	Devices	Other external devices, peripherals	open, close, read, write
13	User Processes	User process	fork, quit, kill, suspend, resume
14	Extended Types	Extended type objects	create, delete, declare type, invoke operation, verify parameters
15	Shell	User programming environment	statements in shell language

TABLE I: Example of an Operating System Design Hierarchy.

intermachine communications; the task of managing files and other objects across several machines would then fall to the user. If the communications layer were lower than the virtual memory level (e.g., part of the process manager), the virtual address space would span the memories of several machines; no efficient address mapping mechanism is known for this case. The middle level is the lowest level at which the communications layer has access to the functions of its host machine necessary to meet its reliability requirements.

Electronic circuits
Instruction set
Procedures
Interrupts
Primitive Processes
Capabilities
Local secondary store
Virtual memory
Communications
Directories
Files
(Pipes)
Devices
User Processes
Extended Types
Shell

TABLE II: Insertion of communications layer.

In other words, placing the communications layer down at the level of primitive processes can be optimal only in a very tightly coupled system in which the main memory units of each machine are part of the same address space. Placing the communications layer at the user level can be optimal only for loosely coupled machines whose operating systems use incompatible schemes for nam-

ing, sharing, and communicating. The placement of the communications layer as in Table II is a compromise permitted by the moderate degree of coupling among machines on a local network.

The following discussion includes specifications of operations implemented in the communications layer and the directory layer. These operations take capabilities as parameters. A capability is a hardware-protected pointer to an object. Its format is shown in Figure 1. The machine number identifies the machine on which the capability was created; if the local bit is set, the capability can be interpreted only on that machine. (If a procedure on one machine is given a capability interpretable on another, it can either reject that capability or ask a process on the other machine to interpret that capability.)

The Communications Layer

The purpose of the communications layer is to provide a single mechanism for exchanging information between two processes, independent of whether they are on the same or different machines.

The external interface presented by the communications layer is suggested by Figure 2, which shows Process 1 sending a sequence of segments to Process 2. The sequence is moved across a *channel*, which is an object created and managed by the communications layer. When the two processes are on the same machine, the queue of segments is in shared memory. The READ and WRITE operations reduce to the familiar "send" and "receive" for message queues [e.g., Brin73]. Figure 3 illustrates what happens when the two processes are on different machines. The communications layer must implement the net-

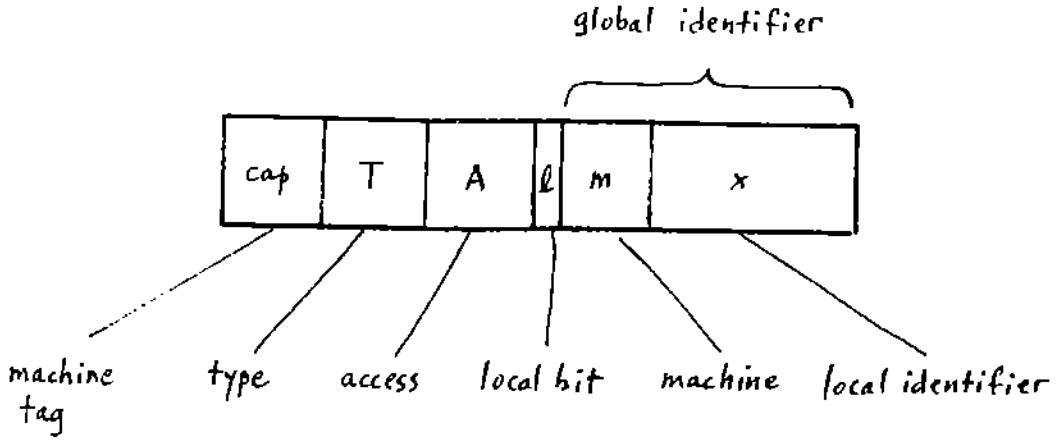

FIGURE 1: Format of a capability for a multi-machine system.

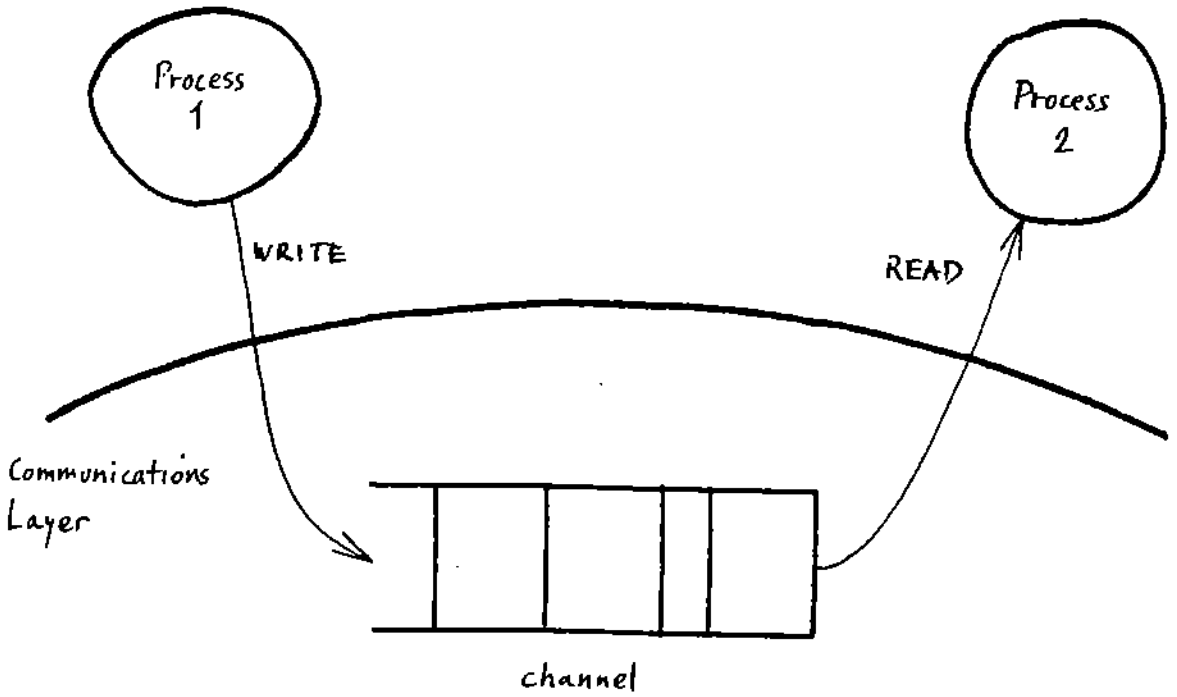

FIGURE 2: User's view of communications layer.

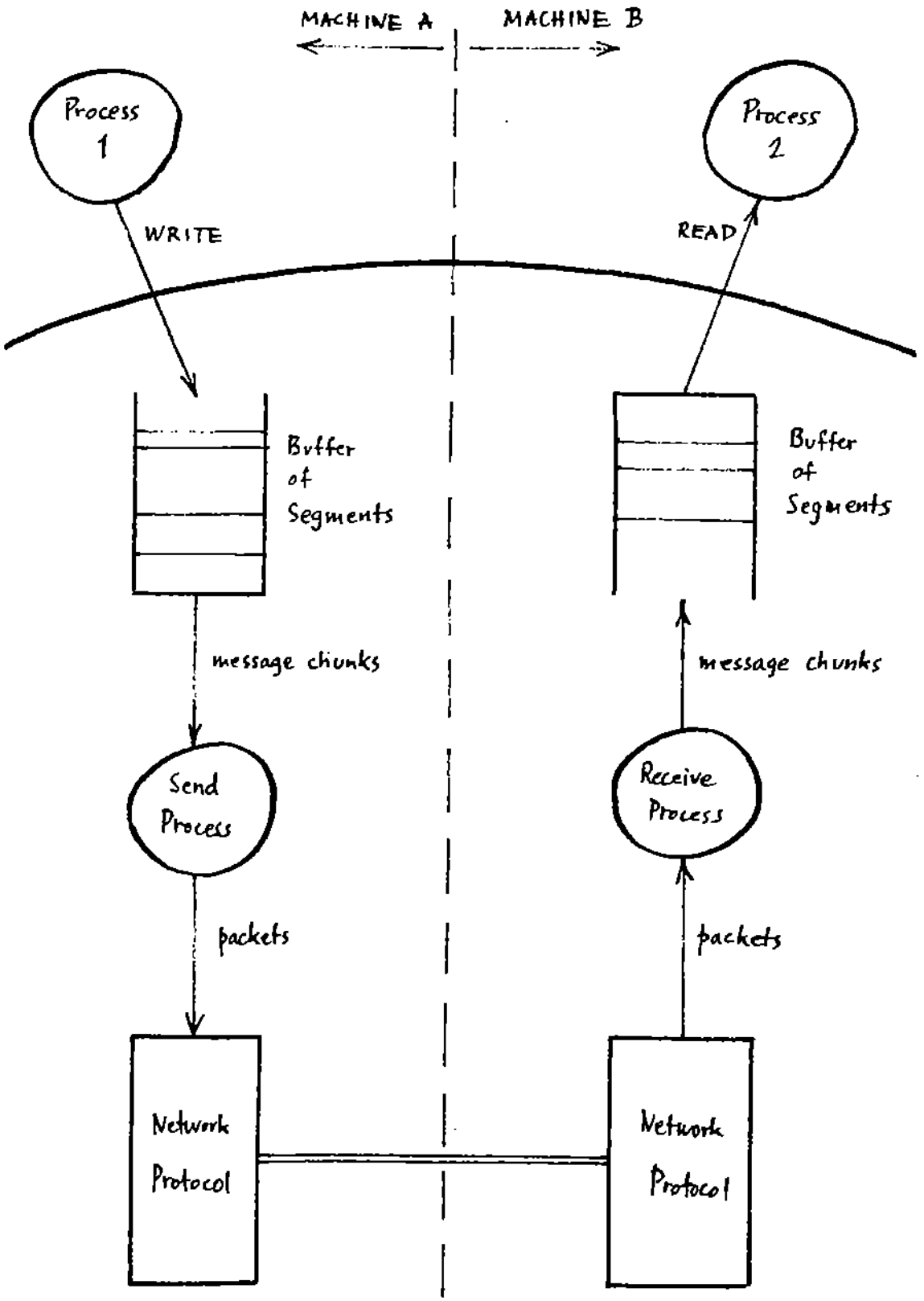

FIGURE 3: Internal view of communications layer.

work protocols required to move information reliably between machines. These protocols must be able to recover from such errors as lost packets, duplicated packets, unacknowledged packets, and packets received out of order. (Compared to long-haul networks protocols, local network protocols may need less mechanism to meet reliability requirements because the error rate is lower.)

Form of call	Effect
ch_cap := CREATE_CH	Creates a new channel and returns a channel capability for it; if the caller then stores this capability in a directory entry, the channel becomes available throughout the system.
DELETE_CH(ch_cap)	Undoes a create channel operation.
op_ch_cap := OPEN(ch_cap, rw)	Opens the channel named by the channel capability; returns an open channel capability with write permission enabled if rw="write" and read permission enabled if rw="read". (Fails if the channel is already open for writing when rw="write" or reading when rw="read".) If both sender and receiver are on the same machine, the open channel control block will indicate that segments can be transferred directly from sender to receiver.
CLOSE(op_ch_cap)	Undoes the open channel operation. (Uses the write/read permission bits in the capability determine which open operation to undo.)
WRITE(seg_cap, op_ch_cap)	Causes the segment named by the given segment capability to be transmitted over the given open channel. (Fails if the open channel capability does not contain write permission.)
seg_cap := READ(op_ch_cap)	Waits until there is a segment in the channel to receive, then returns a capability pointing to it. (Fails if the open channel capability does not contain read permission.)

TABLE III: Example Specification of Communication Layer Interface.

Specifications for six channel operations are outlined in Table III. There are commands to create and delete channels. A channel capability can be passed to another machine (over an existing open channel) for later use by a companion process on that machine; a channel capability can also be listed in the directory hierarchy, whereupon the channel becomes accessible throughout the system. There are commands to open and close channels: the sender and receiver must each open the channel; at most one sender and one receiver are allowed. If the sender and receiver are on different machines, a connection protocol must assure the consistency of the two open channel control blocks. And there are read and write commands for moving a segment of information across the channel.

Because channels have the same properties as UNIX pipes, it is no longer necessary to include a pipe manager level. (Hence, this level is shown in parentheses in Table II.) Channels are like ports in the Intel 432 [Kahn81].

Because user processes are defined at a higher level, it is not possible to open a connection directly to another user process. It is also not normally possible to open a connection directly to a primitive process: they are short-lived and not listed in the directory hierarchy.

It is possible to store a capability in a segment and send that segment over a channel to another machine. If the capability is of a type that can only be interpreted locally (i.e., on the machine that issued it), the other machines must refuse to interpret it. For example, a segment or open-channel capability defined on one machine does not map to a meaningful object on another machine. This requirement is easy to implement in a machine's hardware because capabilities contain the identifier of the issuing machine and a local bit

(see Figure 1); the READ and WRITE operations do not check for capabilities in the segments sent over channels.

If the design were altered to allow global segments, channels would still be necessary: a communications layer would be required for reliable updating of directories on all machines and for synchronizing a sender and receiver. (The directory update problem will be described in the next section.)

The Directory Layer

The purpose of the directory layer is to implement a systemwide directory structure that permits tree pathnames to be used as global names for any permanent object. Figure 4 shows that each entry of a directory contains Name, Access, and Capability fields for each object listed. A directory containing only the self and parent entries is considered "empty".

Only capabilities for permanent objects may be placed in a directory. In the hierarchy of Table 1, this includes channels, directories, files, devices, user processes, and extended types; it excludes capabilities for segments, open channels, open files, and open devices, which have meaning only on the machines that issued them. Information about object attributes, such as ownership or time of last use, is kept in the object descriptor blocks maintained by the object manager levels.

The directory layer simply stores global capabilities but does not attempt to interpret them. The responsibility for mapping a capability to an object lies with the level that manages that type of object. The directory level cannot locate any object except a directory.

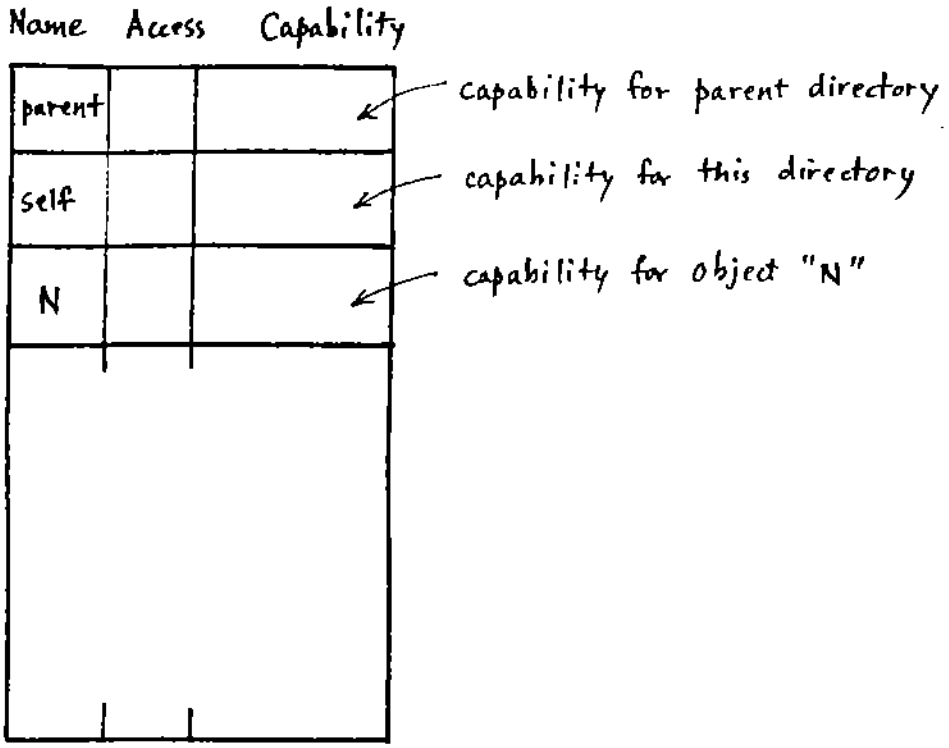

FIGURE 4: Format of a directory.

The directory layer also has the responsibility for ensuring that the directory hierarchy is consistent across all machines of the system. This can be accomplished by methods for replication in a distributed database system [Chu75, Chu76, Seli80]. Any operation that modifies a nonlocal directory must broadcast the change to update-processes in the directory levels of the other machines. (The communications layer is used for this purpose.) To control the number of update messages in a large system, the full directory tree would be kept on only a small subset of machines (e.g., two or three) implementing a "stable store". Copies of the portions of the directory structure being accessed a given user can be encached in a workstation after that user logs in. Updates need be sent only to the stable-store machines and thence to affected workstations.

An example specification of the external operations of a directory layer is given in Table IV. The specifications allow higher levels to create objects and store capabilities for them in directories. The ATTACH operation is used to enter an object capability into a directory under a given name; the DETACH operation undoes this. Both these operations must notify the update processes on other machines so that (nonlocal) changes become effective throughout the system. Unattached objects will not be retained after termination of the user process that created them.

The ATTACH operation allows its caller to specify an access code that will apply to this entry and may reduce privileges enabled in the capability's access field. An access code can be detailed, like Access Control Lists in Multics, or simple, like owner-group-public bits in UNIX. The access field of the capability returned by a SEARCH operation will be the AND of the access code pertaining to

Form of Call	Effect
dir_cap := CREATE_DIR(access)	Allocate an empty directory with its permission bits set to the given access code. Return a capability for it. (This directory is not attached to the directory tree.)
DELETE_DIR(dir_cap)	Remove the given directory. (Fails if the directory is nonempty.)
ATTACH(obj_cap, dir_cap, access, name)	Make an entry of the given name in the given directory; store the given object capability and given access code in it. (Fails if the name already exists in the given directory or if the object is a directory whose parent is defined.) If the given directory is nonlocal, notify the update processes in other machines of the new entry for the given object. If the given object is a local directory: mark it as nonlocal and notify the update processes on other machines of the entire subtree rooted at this directory.
DETACH(dir_cap, name)	Remove the entry of the given name from the given directory. (Fails if the name does not exist in the given directory of if the named object is a nonempty directory.) If the given directory is nonlocal, notify the update processes on other machines.
obj_cap := SEARCH(dir_cap, name)	Find the entry of the given name in the given directory and return a copy of the capability stored therein. Set the access field in this capability to the minimum privilege enabled by the access fields of the directory entry and the capability. (Fails if the name does not exist in the given directory.)
seg_cap := LIST(dir_cap)	Return a copy of the contents of the directory. (A user-level program can interrogate the other levels for other information about the objects listed in the directory -- e.g., date of last change.)

TABLE IV: Example Specification of a Directory Manager Interface.

the owner of its caller and the access field already in the capability.

The ATTACH and DETACH operations are more complex when applied to directories. On creation, a directory capability is "local" and can be

interpreted only on the creating machine. ATTACH operations on local directories do not notify other machines. When a local directory is attached to a nonlocal directory, ATTACH must traverse the entire (local) subtree rooted at the (local) directory and notify the other machines; in so doing it must convert directory capabilities contained therein to nonlocal form.[3] The ATTACH operation must also define the parent of the newly attached directory; ATTACH fails if a parent is already defined.

The DETACH operation only removes entries from directories; it does not delete the object to which the capability points. To delete an object, the DELETE operation of the level that manages that type of object must be used. To minimize inadvertant deletions, DETACH and DELETE operations fail if applied to nonempty directories.

The SEARCH operation returns the capability stored with a given name in a directory; a search usually precedes other operations on an object, e.g., opening and reading a file. The LIST operation provides the raw data used by a formatting program to prepare a summary of the objects listed in a directory.

The specifications in Table IV are not intended to be a complete set of operations for a directory manager. (For example, we specified no command to change the access field in a directory entry.) The purpose is to illustrate the possibility of replicating the directory structure consistently among several machines.

---

[3] This strategy incurs no additional cost relative to a strategy that notifies other machines as each entry is made in a directory; the same set of notifications must be issued sooner or later.

Higher Levels

Each level above the directory level still has some responsibility for hiding the physical locations of objects it manages. The hiding of higher-level objects is not completely achieved by the communications and directory layers.

Form of Call	Effect
op_t_cap := OPEN(t_cap, rw)	Open a connection to the object of type t for reading (if rw="read") or writing (if rw="write"). Return a local capability pointing to the open connection. The access code in the open connection capability is set to the value of rw.
CLOSE(op_t_cap)	Close the connection specified by the given open connection capability.
seg_cap := READ(op_t_cap)	Store a copy of the state of the given, open object in a segment and return a capability for it. (Fails if the given open connection capability does not enable reading.)
WRITE(op_t_cap, seg_cap)	Set the state of the give open object to the value contained in the given segment. (Fails if the given open connection capability does not enable writing.)

TABLE V: Example Specification of a Generic Channel, File, and Device Interface.

Table V illustrates the specification of a generic interface for channels, files, and devices. This interface allows these three types of object to be connected interchangeably to any process. This interface is completed at the devices level, which finishes the task of hiding the physical locations of files and devices.

Suppose a process opens a connection to a file located on a different machine. What happens? There are two alternatives:

1. Open a pair of channels to a process on the file's home machine; the read and write command are relayed via the forward channel to that surrogate process for remote execution; results are passed back over the reverse channel.
2. Move the file from its current machine to the machine on which the file is being opened; thereafter all read and write operations are local.

Both methods are feasible. An experimental version of the first is the Berkeley Version 4.2 UNIX system [Joy82]. An experimental version of the second is the Purdue STORK file system [Pari83].

The device manager level must hide the fact that each device is physically attached to a given machine. If each device's driver is encapsulated in a special process, the OPEN operation can set up a channel to the driver process for that device.

The shell may also have to hide machine dependence. For example, the shell can fork each user process of a pipeline on a different machine to increase parallelism. Although these processes communicate by channels, the shell must determine a machine for each.

Because the open-connection capability is local, the READ and WRITE commands do not have to deal with the problem of finding a nonlocal object. Only the OPEN command need solve this problem.

CONCLUSION

Distributed systems can be hidden in the sense that users need not be concerned with any resource allocation decisions that depend on the physical characteristics of the system. The key to this goal is the layered operating system: above some level (the communications layer) the fact that objects are on different machines is irrelevant. The directory layer can maintain a common name space, defined by the directory hierarchy, for permanent objects across all machines of the system. Each of the higher levels, in turn, removes the remaining vestiges of the physical locations of the objects it manages.

Long-haul network services, such as remote login and file transfer, need not be visible in the user environment; they can be hidden in the communications layer. At the user level they are superceded by operations that permit any authorized process access to any global object in the system. Explicit file (and object) transfer within the distributed system is not needed because all objects belong to a common name space, defined by the directory hierarchy.

Two systems connected by a long-haul net can be amalgamated into a single distributed system if they both use the same layered operating system design. The communications layer can include protocols for the long-haul net, and the file system layer can maintain consistency with the directory hierarchy on the distant machine. If the response time between the two systems is too high for this to be feasible, the communications layer can still be used to open a connection between two file transfer processes on the two systems.

The design can handle heterogeneous systems containing special purpose machines such as file servers, stable stores, or supercomputers. The local

operating system of a special-purpose machine can be very simple because it does not require all the functions of a machine supporting a user programming environment. Such an operating system needs only to receive requests and return responses over the network; it allocates local resources among pending requests.

Most of the lower layers (as high as capability management) are sufficiently well understood that they can appear mainly in hardware and microcode. Layers for communications, directories, files, devices, user processes, extended type objects, and shells are not difficult to design; the concepts embodied in them are well understood (e.g., in UNIX [Ritc74] or PSOS [Neum80]). Thus it is reasonable to suppose that a system having the properties described here can easily be fit into some of the new workstations now coming to market. It follows that the goals of hiding the network and achieving functional redundancy -- can be attained.

ACKNOWLEDGEMENTS

Doug Comer has also argued that the optimal position of the communications layer in the design hierarchy is determined by the degree of coupling among the machines of the system. Jim Bouhana and Walter Tichy have been very helpful in specifying the levels of the design hierarchy. Karl Levitt, Peter Neumann, and Michael Meliar-Smith have, in many discussions, provided many insights into level structured operating systems, especially PSOS.

REFERENCES

Bart81. Bartlett, J., "A NonStop(TM) Kernel," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), 22-29.

Birr82. Birrell, A. D., R. Levin, R. M. Needham, and M. D. Schroder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* 25(4) (April 1982), 260-274.

Brin78. Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM* 21(11) (November 1978), 934-941.

Brin73. Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ (1973).

Chu75. Chu, W. W. and E. E. Nahouraii, "File Directory Design Considerations for Distributed Databases," *International Conference on Very Large Databases*, (1975), 543-545.

Chu76. Chu, W. W., "Performance of File Directory Systems for Databases in Star and Distributed Networks," *Proceedings of the National Computer Conference*, (1976), 577-587.

Cox81. Cox, G. W., W. M. Corwin, K. K. Lai, and F. J. Pollack, "A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), 125-126.

Denn81. Denning, P. J., T. Don Dennis, and J. A. Brumfield, "Low Contention Semaphores and Ready Lists," *Communications of the ACM* 24(10) (October 1981), 687-699.

Denn66. Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM* 9(3) (March 1966), 143-155.

Dijk68. Dijkstra, E. W., "The Structure of the THE-Multiprogramming System," *Communications of the ACM* 11(5) (May 1968), 341-346.

Hoar78. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* 21[B(8) (August, 1978), 666-677.

Intc81. Intel, *Introduction to the iAPX 432 Architecture*, Intel Corporation, Santa Clara, CA (1981).

Jone79. Jones, A. K., R. J. Chansler Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, A Multiprocessor Operating System for the Support of

Task Forces," *Proceedings of the Seventh Symposium on Operating Systems Principles*, (December 1979), 117-127.

Joy82. Joy, W., E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," CSRG Manual, University of California, Department of EECS, Berkeley, CA (September 1982).

Kahn81. Kahn, K. C., W. M. Corwin, T. Don Dennis, H. D'Hooge, D. E. Hubka, L. A. Hutchins, J. T. Montague, F. J. Pollack, and M. R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), 127-136.

Neum80. Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson, "A Provably Secure Operating System, its Applications, and Proofs," CSL-116 (2nd edition), SRI International, Menlo Park, CA (May 7, 1980).

Ousl80. Ousterhout, J. K., D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications of the ACM* 23(2) (February 1980), 92-105.

Pari83. Paris, J.-F. and W. F. Tichy, "STORK: An Experimental File System for Computer Networks Based on Migration," *IEEE INFOCOM*, (1983), (to appear)

Poll81. Pollack, F. J., K. C. Kahn, and R. M. Wilkinson, "The iMAX-432 Object Filing System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), 127-147.

PopW81. Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, (December 1981), 169-177.

Post81. Postel, J., "DARPA Internet Program Protocol Specifications," RFCs 790-796, USC Information Sciences Institute, Marina del Rey, CA (1981).

Ritc74. Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* 17(7) (July 1974), 365-375.

Seli80. Selinger, P. G., "Replicated Data," pp. 223-231 in *Distributed Data Bases* (F. Poole, Ed.), Cambridge University Press, Cambridge, England (1980).

Smit82. Smith, D. C., C. Irby, R. Kimball, and E. Harslem, "The Star User Interface: An Overview," *Proceedings of the AFIPS National Computer*

Conference, (1982), 515-528.

Swan77. Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "Cm* A Modular Multiprocessor," *Proceedings of the National Computer Conference*, (June 1977), 636-644.

Tane81. Tanenbaum, A. S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ (1981).

Wilk79. Wilkes, M. V. and D. J. Wheeler, "The Cambridge Digital Communication Ring," *Proceedings Local Area Communication Network Symposium*, (May 1979).

Wulf81. Wulf, W. A., R. Levin, and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill (1981).